

REPORT DOCUMENTATION PA

AD-A257 072

Public reporting burden for this collection of information is estimated to average 1 hour per response, including gathering the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Project Director, Washington, DC 20503.



gathering
including
Arlington, VA

1. AGENCY USE (Leave blank)

2. REPORT

Final: 27 July 1992

2

4. TITLE AND

Validation Summary Report: Alys, AlysCOMP_069, Version 1.83, Control Data 4336 under TX/IX 1.0.2 (Host & Target), 92073011.11262

5. FUNDING

6.

IABG-AVF

Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND

IABG-AVF, Industrieanlagen-Betriebsgesellschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING
ORGANIZATION

IABG-VSR 108

9. SPONSORING/MONITORING AGENCY NAME(S) AND

Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING
AGENCY

11. SUPPLEMENTARY

DTIC
SELECTE
OCT 20 1992
S D

12a. DISTRIBUTION/AVAILABILITY

Approved for public release; distribution unlimited.

12b. DISTRIBUTION

13. (Maximum 200)

Alys, AlysCOMP_069, Version 1.83, Control Data 4336 under TX/IX 1.0.2 (Host & Target), ACVC 1.11.

92 10 10 010

391490

92-27406



14. SUBJECT

Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A,

15. NUMBER OF

16. PRICE

17. SECURITY
CLASSIFICATION
UNCLASSIFIED

18. SECURITY
UNCLASSIFIED

19. SECURITY
CLASSIFICATION
UNCLASSIFIED

20. LIMITATION OF

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 920730I1.11262
Alsys
AlsyCOMP_069 Version 1.83
Control Data 4336 under TC/IX 1.0.2
Host and Target

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on July 30, 1992.

Compiler Name and Version: AlsysCOMP_069 Version 1.83

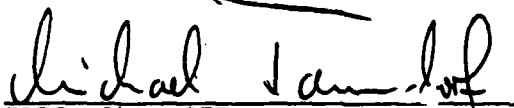
Host Computer System: Control Data 4336 under TC/IX 1.0.2

Target Computer System: Same as Host

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #920730I1.11262 is awarded to Alsys. This certificate expires 24 months after ANSI approval of ANSI/MIL-STD 1815B.

This report has been reviewed and is approved.



IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany



Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311



Ada Joint Program Office
Dr. John Solomon, Director
Department of Defense
Washington DC 20301

DTIC QUALITY CONTROL RECORD 1

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Alsys GmbH & Co. KG

Ada Validation Facility: IABG mbH, Abt. ITE

ACVC Version: 1.11

Ada Implementation:

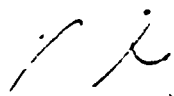
Ada Compiler Name and Version: AlsyCOMP_069, Version 1.83

Host Computer System: Control Data 4336 under TC/IX 1.0.2

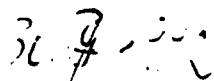
Target Computer System: Control Data 4336 under TC/IX 1.0.2

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.



Customer Signature



Date

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	REFERENCES	1-2
1.3	ACVC TEST CLASSES	1-2
1.4	DEFINITION OF TERMS	1-3
CHAPTER 2	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS	2-1
2.2	INAPPLICABLE TESTS	2-1
2.3	TEST MODIFICATIONS	2-4
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT	3-1
3.2	SUMMARY OF TEST RESULTS	3-1
3.3	TEST EXECUTION	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint
Program Office, August 1990.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is August 02, 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests) (*)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

The following 20 tests check for the predefined type `LONG_INTEGER`; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

C41401A checks that `CONSTRAINT_ERROR` is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception. (See Section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, `MAX_MANTISSA` is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

B86001Y uses the name of a predefined fixed-point type other than type `DURATION`; for this implementation, there is no such type.

C96005B uses values of type `DURATION`'s base type that are outside the range of type `DURATION`; for this implementation, the ranges are the same.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107C..D (2 tests), CE2107H, and CE2107L apply function NAME to temporary sequential, direct, and text files in an attempt to associate multiple internal files with the same external file; USE_ERROR is raised because temporary files have no name.

CE2108B, CE2108D, and CE3112B use the names of temporary sequential, direct, and text files that were created in other tests in order to check that the temporary files are not accessible after the completion of those tests; for this implementation, temporary files have no name.

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

EE2401D uses instantiations of `DIRECT_IO` with unconstrained array and record types; this implementation raises `USE_ERROR` on the attempt to create a file of such types.

CE2403A checks that `WRITE` raises `USE_ERROR` if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3111B and CE3115A associate multiple internal text files with the same external file and attempt to read from one file what was written to the other, which is assumed to be immediately available; this implementation buffers output. (See section 2.3.)

CE3202A expects that function `NAME` can be applied to the standard input and output files; in this implementation these files have no names, and `USE_ERROR` is raised. (See section 2.3.)

CE3304A checks that `SET_LINE_LENGTH` and `SET_PAGE_LENGTH` raise `USE_ERROR` if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that `PAGE` raises `LAYOUT_ERROR` when the value of the page number exceeds `COUNT'LAST`; for this implementation, the value of `COUNT'LAST` is greater than 150000, making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 23 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises `CONSTRAINT_ERROR` when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from Report.Failed was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises `CONSTRAINT_ERROR` when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165, & 175.

C64103A was graded passed by Evaluation Modification as directed by the AVO. This test checks that exceptions are raised when actual parameter values, which result from an explicit type conversion, do not belong to the formal parameter's base type. However this implementation recognizes that the formal parameter is not used within the procedure and therefore the type conversion (and subtype check) need not be made (as allowed by [Ada 83] 11.6.7) and the subsequent expected exception need not be raised. The AVO ruled that the implementation's behavior should be graded passed, given that Report.Failed was invoked only from procedure calls at lines 91 (invoking line 76) and 119 (invoking line 115), yielding the following output:

```
"EXCEPTION NOT RAISED BEFORE CALL -P2 (A)"
"EXCEPTION NOT RAISED BEFORE CALL -P3 (A)"
```

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

CE3111B and CE3115A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests assume that output from one internal file is unbuffered and may be immediately read by another file that shares the same external file. This implementation raises END_ERROR on the attempts to read at lines 87 and 101, respectively.

CE3202A was graded inapplicable by Evaluation Modification as directed by the AVO. This test applies function NAME to the standard input file, which in this implementation has no name; USE_ERROR is raised but not handled, so the test is aborted. The AVO ruled that this behavior is acceptable pending any resolution of the issue by the ARG.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact in Germany for technical and sales information about this Ada implementation system, see:

Alsys GmbH & Co. KG
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

For a point of contact outside Germany for technical and sales information about this Ada implementation system, see:

Alsys Inc.
67 South Bedford Str.
Burlington MA
01803-5152
USA
Tel. +617 270 0030

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a

floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3787	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	87	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	288	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A Magnetic Data Cartridge containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the tape were directly loaded onto the host computer.

After the test files were loaded, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Tests were compiled using the command

```
ada.c 'file name'
```

and linked using the command

```
ada.link -o 'file name' 'main unit'.
```

The option -o was used to assign a dedicated file name to the generated executable image.

Class B tests, the executable not applicable tests, and the executable tests of class E were compiled using the full listing option -l. For several tests, compiler listings were added and concatenated using the option -L 'file name'. The compiler is described in Appendix B, compilation system options, chapter 4.2 of the User Manual on page 40.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Data Cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	255 -- Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	"" & (1..V/2 => 'A') & ""
\$BIG_STRING2	"" & (1..V-1-V/2 => 'A') & '1' & ""
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	"" & (1..V-2 => 'A') & ""

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	2_147_483_647
\$DEFAULT_MEM_SIZE	2147483648
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	MIPS_TCIX
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGUSR1)
\$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGUSR2)
\$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGALRM)
\$FIELD_LAST	512
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	" "
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	0.0
\$GREATER_THAN_DURATION_BASE_LAST	200_000.0
\$GREATER_THAN_FLOAT_BASE_LAST	16#1.0#E+32
\$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.8#E+32
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0
\$HIGH_PRIORITY	15

MACRO PARAMETERS

```

$ILLEGAL_EXTERNAL_FILE_NAME1
    /nodir/file1

$ILLEGAL_EXTERNAL_FILE_NAME2
    /wrongdir/file2

$INAPPROPRIATE_LINE_LENGTH
    -1

$INAPPROPRIATE_PAGE_LENGTH
    -1

$INCLUDE_PRAGMA1      PRAGMA INCLUDE ("A28006D1.ADA")
$INCLUDE_PRAGMA2      PRAGMA INCLUDE ("B28006D1.ADA")

$INTEGER_FIRST        -2147483648
$INTEGER_LAST         2147483648
$INTEGER_LAST_PLUS_1  2147483648

$INTERFACE_LANGUAGE   C

$LESS_THAN_DURATION   -0.0

$LESS_THAN_DURATION_BASE_FIRST
    -200_000.0

$LINE_TERMINATOR      ASCII.LF

$LOW_PRIORITY         0

$MACHINE_CODE_STATEMENT
    NULL;

$MACHINE_CODE_TYPE     NO_SUCH_TYPE

$MANTISSA_DOC         31

$MAX_DIGITS           15

$MAX_INT              2147483647

$MAX_INT_PLUS_1       2_147_483_648

$MIN_INT              -2147483648

$NAME                 SHORT_SHORT_INTEGER

$NAME_LIST            MIPS_TCIX

$NAME_SPECIFICATION1   /ben0/c183/acvc11/chape/X2120A
$NAME_SPECIFICATION2   /ben0/c183/acvc11/chape/X2120B

```

MACRO PARAMETERS

\$NAME_SPECIFICATION3	/ben0/c183/acvc11/chape/X3119A
\$NEG_BASED_INT	16#FFFFFFFFE#
\$NEW_MEM_SIZE	2147483648
\$NEW_SYS_NAME	MIPS_TCIX
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	NEW INTEGER
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	10240
\$TICK	1.0/3686400.0
\$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
\$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
\$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed.

§4.1 and Chapter 5 describe in detail how to call the Compiler, the Completer, which is called to generate code for instances of generic units, and the Linker.

Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.

The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.

Finally, the log of a sample session is given in Chapter 7.

The following conventions are used:

<ADA_dir> stands for the directory where the Alsys Ada System is located on your computer. This directory can be found using the `ada.lib` command. (If you have no library just create one with the `ada.create` command.) The output of the `ada.lib` command displays the directory in the first line.

4.1 Compiling Ada Units

The command `ada.c` invokes the Compiler, and optionally Completer and Linker of the Alsys Ada System.

<code>ada.c</code>	Command Description
--------------------	---------------------

NAME

`ada.c` – Alsys Ada System compile command

SYNOPSIS

`ada.c` [option ...] [file ...] [`-ld ldopt`]

DESCRIPTION

Compilation, Completion and Linking are performed in that order. The Completer is called if the `-C` or the `-m` option is specified. The Linker is called if the `-m` option is specified. By default, only the compiler runs and compiles the source(s) in the given *files*.

The source file may contain a sequence of compilation units (cf. LRM(§10.1)). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §4.4). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The command delivers a non-zero status code on termination (cf. `exit(2)`) if one of the compilation units contained errors.

file specifies the file(s) to be compiled. The maximum length of lines in *file* is 255. The maximum number of source lines in *file* is 65534.

Note: If you specify a file name pattern, which is replaced by one or more file names by the shell, the order of the compilation is alphabetical, which is not always successful. Thus file name patterns should be used together with the option `-a`. With this option the sources can be processed in any order.

The generation of listing output is controlled by options `-l` and `-L`. The default listing filename for a compilation is the basename, cf. `basename(1)`, of the source file with suffix `.l`; when the source file already has a suffix, it is replaced by the suffix `.l`. When an automatic recompilation is performed through option `-R` the basename is taken from the original source file name stored in the library.

-A Controls whether automatic inline expansion is performed. A subprogram *S* is automatically inlined at a place *P* where *S* is called, if the following conditions hold: *S* meets the requirements for explicit inlining via `PRAGMA inline` (cf. §15.1.1); subprogram specification and subprogram body of *S* are in the same compilation unit; and the estimated code size of *S* when expanded inline is not greater (or only slightly greater) than the call it replaces. (The estimation of size is based on heuristics and is not exact; however, it is designed to give a close approximation.) If you specify `-A`, automatic inline expansion is suppressed.

By default, automatic inline expansion is performed.

-a Specifies that the Compiler only performs syntactical analysis and the analysis of the dependencies on other units. The units in *file* are entered into the library if they are syntactically correct. The actual compilation is done later.

Note: An already existing unit with the same name as the new one is replaced and all dependent units become obsolete, unless the source file of both are identical. In this case the library is *not* updated because the dependencies are already known.

By default, the normal, full compilation is done.

-C *unitlist* Requests the completion of the units in *unitlist*, which is a white space separated list of unit names. *unitlist* must be a single shell argument and must therefore be quoted when it has more than one item. Example with two units:

```
ada.c -C "our_unit my_unit"
```

The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).

If a listing is requested the default filename used is *complete.l*. The listing file contains the listing information for all units given in *unitlist*.

-c Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need normally not be known by the user. The Recompiler and the Debugger know this name. You can use the *ada.list -l* command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.

If **-c** is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.

A copy of the source is only taken if the *ada.c* command results in a successful compilation (and the option **-c** is not given). In particular, no copy is taken if the option **-a** is given since in this case no compilations are carried out.

-D By default debug information for the Alsys Ada Debugger is generated and included in the executable file. When the **-D** option is present, debug information is not included in the object file. If the program is to run under the control of the Debugger it must be linked without the **-D** option.

-
- I** Controls whether inline expansion is performed as requested by `PRAGMA inline`. If you specify `-I` these pragmas are ignored.
- By default, inline expansion is performed.
- l** Generates listing files with default filenames (see above) in the current directory (use option `-L` for redirecting to another directory).
- L *directory*** Generates listing files with default filenames (see above) in directory *directory*.
- L *file*** Concatenates all listings onto file *file*.
- ld *ldopt*** This option can be used to supply options for the call of `ld(1)` when linking a program by the `-m` option. `-ld` followed by the options to be passed to `ld(1)` must be the last items of the command.
- m *unit*** Specifies the name of a main program, which must be a parameterless procedure. This option will cause the completion of any generic instantiations in the program; if a listing is requested, the listing options have the same meaning as for the complete option; if the completer has already been called by the `-C` option, the listing output is appended to that completer listing file. If all compilations are successful, the linker is invoked to build an executable file; if a listing is requested, the default filename for the linker listing is `link.1`.
- O*l*** Restricts optimizations to level *l*. Level 0 indicates no optimizations, level 1 indicates partial optimizations, level 2 indicates full optimization. Default is full optimization.
- Partial optimizations allows those optimizations that do not move code globally. These are: Constant propagation, copy propagation, algebraic simplifications, runtime check elimination, dead code elimination, peephole and pipeline optimizations. This optimization level allows easier debugging while maintaining a reasonable code quality.
- Full optimization enforces the following optimizations in addition to those done with `-O1`: Global common subexpression elimination and keeping local variables in registers.
- o *file*** When linking is requested by `-m` this option can be used to specify the name of the generated executable file. By default, the unit name given with the `-m` option is used; this value is taken literally, i.e. upper and lower case letters are distinguished.

-
- | | |
|-------------------|--|
| -R | Indicates that a recompilation of a previously analyzed source is to be performed. This option should only be used in commands produced by the <code>ada.make</code> command. |
| -r | Suppresses the generation of an executable file when linking is requested. See the <code>-r</code> option of the <code>ada.link</code> command (§5) for details. |
| -S | Controls whether all run-time checks are suppressed. If you specify <code>-S</code> this is equivalent to the use of <code>PRAGMA suppress</code> for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where <code>PRAGMA suppress_all</code> appears in the source. |
| -s | Controls whether machine code is appended to the listing file. <code>-s</code> has no effect if no listing is requested or <code>-a</code> (analyze only) is specified.

By default, no machine code is appended to the listing file. |
| -t | Suppresses selective linking. Selective linking means that only the code of those subprograms which can actually be called is included in the executable image. With <code>-t</code> the code of all subprograms of all packages in the execution closure of the main procedure is linked into the executable image.

Note: The code of the runtime system and of the predefined units is always linked selectively. |
| -v | Controls whether the <code>ada.c</code> command writes additional information onto standard error.
By default, no additional information is written. |
| -y <i>library</i> | Specifies the program library the <code>ada.c</code> command works on. It needs write access to the library.
The default library is <code>adalib</code> . |
-

End of Command Description

4.2 Completing Generic Instances

Since the Compiler does not generate code for instances of generic bodies, the Completer must be used to complete such units before a program using the instances can be executed. The Completer must also be used to complete packages in the program which do not require a body. This is done implicitly when the Linker is called.

It is also possible to call the Completer explicitly with the `-C` option of the `ada.c` command.

4.3 Automatic Compilation

The Alsys Ada System offers three different kinds of automatic compilation. It supports

- automatic recompilation of obsolete units
- automatic compilation of modified sources
- automatic compilation of new sources with unknown dependencies

In the following the term *recompilation* refers to the recompilation of an obsolete unit using the identical source which was used the last time. (This kind of recompilation could alternatively be implemented by using some appropriate intermediate representation of the obsolete unit.) This definition is stronger than that of the LRM (10.3). If a new version of the source of a unit is compiled we call it *compilation*, not a *recompilation*.

The set of units to be checked for recompilation or new compilation is described by specifying one or more units and the kind of a closure which is to be built on them. In many cases you will simply specify your main program.

The automatic compilation of modified sources is the default effect of the `ada.make` command. It determines the set of modified sources and generates and executes a command file for calling the Compiler in an appropriate order. Optionally, obsolete units can be recompiled subsequently.

If only the recompilation of obsolete units is desired, this can be specified by the `-Rr` option of the `ada.make` command. It determines the set of obsolete units and generates and executes a command file for calling the Compiler in an appropriate order.

The recompilation is performed using the copy of the obsolete units which is (by default) stored in the library. (If the user does not want to hold a copy of the sources, the `ada.make` command can use the original source.)

5 Linking

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

The TC/IX system linker is used by the Alsys Ada Linker.

To link a program, call the `ada.link` command. The Linker can also be called directly from the `ada.c` command and from the `ada.make` command.

ada.link	Command Description
-----------------	----------------------------

NAME

`ada.link` – invoke the Alsys Ada System linker

SYNOPSIS

`ada.link [option ...] unit [-ld ldopt]`

DESCRIPTION

The `ada.link` command invokes the Alsys Ada Linker.

The Linker builds an executable file. The default file name of the executable file is the unit name of the main program given with the *unit* parameter. This value is taken literally, i.e. upper and lower case letters are distinguished.

unit specifies the library unit which is the main program. This must be a parameterless library procedure.

- | | |
|----|---|
| -A | This option is passed to the implicitly invoked Completer. See the same option with the <code>ada.c</code> command. |
| -c | Suppresses invocation of the Completer of the Alsys Ada System before the linking is performed. Only specify -c if you are sure that there are no instantiations or implicit package bodies to be compiled, e.g. if you repeat the <code>ada.link</code> command with different linker options. |

- D** By default debug information for the Alsys Ada Debugger is generated and included in the executable file. When the **-D** option is present, debug information is not included in the object file. If the program is to run under the control of the Debugger it must be linked without the **-D** option.
- I** Controls whether inline expansion is performed as requested by `PRAGMA inline`. If you specify **-I** these pragmas are ignored.
- By default, inline expansion is performed.
- l** If **-l** is specified the Linker of the Alsys Ada System creates a listing file containing a table of symbols which are used for linking the Ada units. This table is helpful when debugging an Ada program with the TC/IX debugger. The default name of the listing file is `link.1`. By default, the Linker does not create a listing file. This option is also passed to the implicitly invoked Completer, which by default generates a listing file `complete.1` if **-l** is given.
- L directory** The listing files are created in directory *directory* instead of in the current directory (default).
- L file** The listing files are concatenated onto file *file*.
- ld ldopt** This option can be used to supply options for the call of `ld(1)`. **-ld** followed by the options to be passed to `ld(1)` must be the last items of the command.
- ol** This option is passed to the implicitly invoked Completer. See the same option with the `ada.c` command.
- o file** Specifies the name of the executable file. The default file name of the executable file is the unit name of the main program.
- r** Suppresses the generation of an executable file. In this case the generated object file contains the code of all compilation units written in Ada and of those object modules of the predefined language environment and of the Ada run time system which are used by the main program; references into the Standard C library remain unresolved. The generated object module is suitable for further `ld(1)` processing. The name of its entry point is `main`.
- S** This option is passed to the implicitly invoked Completer. See the same option with the `ada.c` command.

-
- s** This option is passed to the implicitly invoked Completer. See the same option with the `ada.c` command. If a listing is requested and `-s` is specified, the Linker of the Alsys Ada System generates a listing with the machine code of the program starter in the file `link.1`. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program. By default, no machine code is generated.
- t** Suppresses selective linking. Selective linking means that only the code of those subprograms which can actually be called is included in the executable file. With `-t` the code of all subprograms of all packages in the execution closure of the main procedure is linked into the executable file.
- Note: The code of the runtime system and of the predefined units is always linked selectively, even if `-t` is specified.
- v** Controls whether the `ada.link` command writes additional information onto standard error, and is also passed to the implicitly invoked Completer. By default, no additional information is written.
- y library** Specifies the program library the command works on. The `ada.link` command needs write access to the library unless `-c` is specified. If `-c` is specified the `ada.link` command needs only read access. The default library is `adalib`.
-

End of Command Description

The `ada.link` command implicitly calls the TC/IX System Linker using the command

```
/bsd43/bin/ld [-N ] -o resultfile /bsd43/usr/lib/cmplrs/cc/crt1.o \
    obj rtslib ld_options -lc /bsd43/usr/lib/cmplrs/cc/crtn.o
```

unless the `-r` option is specified. When `-r` is specified, the Linker is called with the command

```
/bsd43/bin/ld [-N] -o resultfile -r obj rtslib ld_options
```

Here, *obj* denotes the file containing the object module which is produced by the Ada Linker and *rtslib* the archive library containing the Ada runtime system. (This may be `librtsdbg.a` resp. `librts.a` if the Alsys Ada Linker is called with option `-D`. In this case the `-N` option is missing.)

If you invoke *ld(1)* by yourself to link the executable object rather than having the Ada Linker doing it automatically, then you must explicitly specify the startup modules (see below) and any libraries you want linked into the Ada program. Furthermore, the option *-N* of *ld(1)* should be specified to allow the resulting object file to be debugged by the Alsys Ada Debugger. (Note that debugging is only possible if the option *-D* was not passed to the Ada Linker.)

The startup module must satisfy the following requirements:

- A global variable called *environ* is defined containing a pointer to the current environment (cf. *environ(7)*.)
- The Ada main program is called using the entry point *main*.
- *argc* and *argv* are passed as arguments to *main*.

Note that instructions following the call of *main* will never be executed.

By default, the Standard C runtime startup routines */bsd43/usr/lib/cmplrs/cc/crt1.o* and */bsd43/usr/lib/cmplrs/cc/crtn.o* are used.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are contained in the following Predefined Language Environment (chapter 13 page 275 ff of the compiler user manual).

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

PACKAGE standard IS

TYPE boolean IS (false, true);

-- The predefined relational operators for this type are as follows:

```
-- FUNCTION "=" (left, right : boolean) RETURN boolean;
-- FUNCTION "/=" (left, right : boolean) RETURN boolean;
-- FUNCTION "<" (left, right : boolean) RETURN boolean;
-- FUNCTION "<=" (left, right : boolean) RETURN boolean;
-- FUNCTION ">" (left, right : boolean) RETURN boolean;
-- FUNCTION ">=" (left, right : boolean) RETURN boolean;
```

-- The predefined logical operators and the predefined logical
-- negation operator are as follows:

```
-- FUNCTION "AND" (left, right : boolean) RETURN boolean;
-- FUNCTION "OR" (left, right : boolean) RETURN boolean;
-- FUNCTION "XOR" (left, right : boolean) RETURN boolean;

-- FUNCTION "NOT" (right : boolean) RETURN boolean;
```

-- The universal type universal_integer is predefined.

TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

-- The predefined operators for this type are as follows:

```
-- FUNCTION "=" (left, right : integer) RETURN boolean;
-- FUNCTION "/=" (left, right : integer) RETURN boolean;
```

```

-- FUNCTION "<" (left, right : integer) RETURN boolean;
-- FUNCTION "<=" (left, right : integer) RETURN boolean;
-- FUNCTION ">" (left, right : integer) RETURN boolean;
-- FUNCTION ">=" (left, right : integer) RETURN boolean;

-- FUNCTION "+" (right : integer) RETURN integer;
-- FUNCTION "-" (right : integer) RETURN integer;
-- FUNCTION "ABS" (right : integer) RETURN integer;

-- FUNCTION "+" (left, right : integer) RETURN integer;
-- FUNCTION "-" (left, right : integer) RETURN integer;
-- FUNCTION "*" (left, right : integer) RETURN integer;
-- FUNCTION "/" (left, right : integer) RETURN integer;
-- FUNCTION "REM" (left, right : integer) RETURN integer;
-- FUNCTION "MOD" (left, right : integer) RETURN integer;

-- FUNCTION "***" (left : integer; right : integer) RETURN integer;

-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such additional types end
-- with INTEGER as in SHORT_INTEGER or LONG_INTEGER. The
-- specification of each operator for the type universal_integer, or
-- for any additional predefined integer type, is obtained by
-- replacing INTEGER by the name of the type in the specification
-- of the corresponding operator of the type INTEGER, except for the
-- right operand of the exponentiating operator.

TYPE short_short_integer IS RANGE - 128 .. 127;

TYPE snort_integer IS RANGE - 32_768 .. 32_767;

-- The universal type universal_real is predefined.

TYPE float IS DIGITS 6 RANGE
    - 16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;
FOR float'size USE 32;

-- The predefined operators for this type are as follows:

-- FUNCTION "=" (left, right : float) RETURN boolean;
-- FUNCTION "/=" (left, right : float) RETURN boolean;
-- FUNCTION "<" (left, right : float) RETURN boolean;
-- FUNCTION "<=" (left, right : float) RETURN boolean;
-- FUNCTION ">" (left, right : float) RETURN boolean;
-- FUNCTION ">=" (left, right : float) RETURN boolean;

-- FUNCTION "+" (right : float) RETURN float;

```

```
-- FUNCTION "-" (right : float) RETURN float;
-- FUNCTION "ABS" (right : float) RETURN float;

-- FUNCTION "+" (left, right : float) RETURN float;
-- FUNCTION "-" (left, right : float) RETURN float;
-- FUNCTION "*" (left, right : float) RETURN float;
-- FUNCTION "/" (left, right : float) RETURN float;

-- FUNCTION "***" (left : float; right : integer) RETURN float;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.
```

```
TYPE long_float IS DIGITS 15 RANGE
    - 16#0.FFFF_FFFF_FFFF_F8#E256 ..
      16#0.FFFF_FFFF_FFFF_F8#E256;
FOR long_float'size USE 64;
```

```
-- In addition, the following operators are predefined for universal
-- types:

-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
--             RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL; right : UNIVERSAL_INTEGER)
--             RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL; right : UNIVERSAL_INTEGER)
--             RETURN UNIVERSAL_REAL;

-- The Type universal_fixed is predefined.
-- The only operators declared for this type are

-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
--             right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
--             right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;

-- The following characters form the standard ASCII character set.
-- Character literals corresponding to control characters are not
-- identifiers.
```

```
TYPE character IS
    (nul, soh, stx, etx, eot, enq, ack, bel,
```

```

bs.   ht.   lf.   vt.   ff.   cr.   so.   si.
dle.  dc1.  dc2.  dc3.  dc4.  nak.  syn.  etb.
can.  em.   sub.  esc.  fs.   gs.   rs.   us.
'.'   '!'   '"'   '#'   '$'   '%'   '&'   '...'
'('   ')'   '*'   '+'   '-'   '-'   '.'   '/'
'0'   '1'   '2'   '3'   '4'   '5'   '6'   '7'
'8'   '9'   ':'   ';'   '<'   '='   '>'   '?'
'A'   'B'   'C'   'D'   'E'   'F'   'G'
'H'   'I'   'J'   'K'   'L'   'M'   'N'   'O'
'P'   'Q'   'R'   'S'   'T'   'U'   'V'   'W'
'X'   'Y'   'Z'   '['   '\'   ']'   '^'   '_'
'a'   'b'   'c'   'd'   'e'   'f'   'g'
'h'   'i'   'j'   'k'   'l'   'm'   'n'   'o'
'p'   'q'   'r'   's'   't'   'u'   'v'   'w'
'x'   'y'   'z'   '{'   '|'   '}'   '~'   del);

```

```

FOR character USE -- 128 ascii CHARACTER SET WITHOUT HOLES
(0, 1, 2, 3, 4, 5, ..., 125, 126, 127);

```

```

-- The predefined operators for the type CHARACTER are the same as
-- for any enumeration type.

```

```

PACKAGE ascii IS

```

```

-- Control characters:

```

```

nul : CONSTANT character := nul;   soh : CONSTANT character := soh;
stx : CONSTANT character := stx;   etx : CONSTANT character := etx;
eot : CONSTANT character := eot;   enq : CONSTANT character := enq;
ack : CONSTANT character := ack;   bel : CONSTANT character := bel;
bs  : CONSTANT character := bs;    ht  : CONSTANT character := ht;
lf  : CONSTANT character := lf;    vt  : CONSTANT character := vt;
ff  : CONSTANT character := ff;    cr  : CONSTANT character := cr;
so  : CONSTANT character := so;    si  : CONSTANT character := si;
dle : CONSTANT character := dle;   dc1 : CONSTANT character := dc1;
dc2 : CONSTANT character := dc2;   dc3 : CONSTANT character := dc3;
dc4 : CONSTANT character := dc4;   nak : CONSTANT character := nak;
syn : CONSTANT character := syn;   etb : CONSTANT character := etb;
can : CONSTANT character := can;   em  : CONSTANT character := em;
sub : CONSTANT character := sub;   esc : CONSTANT character := esc;
fs  : CONSTANT character := fs;    gs  : CONSTANT character := gs;
rs  : CONSTANT character := rs;    us  : CONSTANT character := us;
del : CONSTANT character := del;

```

```

-- Other characters:

```

```

exclam    : CONSTANT character := '!';
quotation : CONSTANT character := '"';
sharp     : CONSTANT character := '#';

```

```

dollar      : CONSTANT character := '$';
percent     : CONSTANT character := '%';
ampersand   : CONSTANT character := '&';
colon       : CONSTANT character := ':';
semicolon   : CONSTANT character := ';';
query       : CONSTANT character := '?';
at_sign     : CONSTANT character := '@';
l_bracket   : CONSTANT character := '[';
back_slash  : CONSTANT character := '\';
r_bracket   : CONSTANT character := ']';
circumflex  : CONSTANT character := '^';
underline   : CONSTANT character := '_';
grave       : CONSTANT character := '`';
l_brace     : CONSTANT character := '{';
bar         : CONSTANT character := '|';
r_brace     : CONSTANT character := '}';
tilde       : CONSTANT character := '~';

```

```
lc_a : CONSTANT character := 'a';
```

```
...
```

```
lc_z : CONSTANT character := 'z';
```

```
END ascii;
```

```
-- Predefined subtypes:
```

```
SUBTYPE natural IS integer RANGE 0 .. integer'last;
```

```
SUBTYPE positive IS integer RANGE 1 .. integer'last;
```

```
-- Predefined string type:
```

```
TYPE string IS ARRAY(positive RANGE <>) OF character;
```

```
PRAGMA byte_pack(string);
```

```
-- The predefined operators for this type are as follows:
```

```
-- FUNCTION "=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "/=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "<" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "<=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION ">" (left, right : string) RETURN boolean;
```

```
-- FUNCTION ">=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "&" (left : string; right : string) RETURN string;
```

```
-- FUNCTION "&" (left : character; right : string) RETURN string;
```

```
-- FUNCTION "&" (left : string; right : character) RETURN string;
```

```
-- FUNCTION "&" (left : character; right : character) RETURN string;

TYPE duration IS DELTA 2#1.0#E-14 RANGE
    - 131_072.0 .. 131_071.999_938_964_843_75;

-- The predefined operators for the type DURATION are the same
-- as for any fixed point type.

-- the predefined exceptions:

constraint_error : EXCEPTION;
numeric_error    : EXCEPTION;
program_error    : EXCEPTION;
storage_error    : EXCEPTION;
tasking_error    : EXCEPTION;

END standard;
```

13.2 Language-Defined Library Units

The following language-defined library units are included in the master library:

- The PACKAGE `system`
- The PACKAGE `calendar`
- The generic PROCEDURE `unchecked_deallocation`
- The generic FUNCTION `unchecked_conversion`
- The PACKAGE `io_exceptions`
- The generic PACKAGE `sequential_io`
- The generic PACKAGE `direct_io`
- The PACKAGE `text_io`
- The PACKAGE `low_level_io`

13.3 Implementation-Defined Library Units

The master library also contains the implementation-defined library units

- The PACKAGE `collection_manager` (§13.3.1)
- The PACKAGE `timing` (§13.3.2)
- The PACKAGE `command_arguments` (§13.3.3)
- The PACKAGE `text_io_extension` (§13.3.4)
- The PACKAGE `global_pipes` (§13.3.5)
- The PACKAGE `execution_time_profile` (§9.2.1)

13.3.1 The Package `COLLECTION_MANAGER`

In addition to unchecked storage deallocation (cf. LRM(§13.10.1)), this implementation provides the generic package `collection_manager`, which has advantages over unchecked deallocation in some applications; e.g. it makes it possible to clear a collection with a single reset operation. See §15.10 for further information on the use of the collection manager and unchecked deallocation.

The package specification is:

GENERIC

```
TYPE elem IS LIMITED PRIVATE;  
TYPE acc IS ACCESS elem;  
PACKAGE collection_manager IS
```

```
TYPE status IS LIMITED PRIVATE;
```

```
PROCEDURE mark (s : OUT status);
```

```
-- Marks the heap of type ACC and  
-- delivers the actual status of this heap.
```

```
PROCEDURE release (s : IN status);
```

```
-- Restore the status s on the collection of ACC.  
-- RELEASE without previous MARK raises CONSTRAINT_ERROR
```

```
PROCEDURE reset;
```

```
-- Deallocate all objects on the heap of ACC
```

PRIVATE

-- private declarations

END collection_manager;

A call of the procedure `release` with an actual parameter `s` causes the storage occupied by those objects of type `acc` which were allocated after the call of `mark` that delivered `s` as result, to be reclaimed. A call of `reset` causes the storage occupied by all objects of type `acc` which have been allocated so far to be reclaimed and cancels the effect of all previous calls of `mark`.

See §15.2.1 for information on static and dynamic collections and the attribute `STORAGE_SIZE`.

13.3.2 The Package `TIMING`

The package `timing` provides a facility for CPU-time measurement. The package specification is:

PACKAGE `timing` IS

 FUNCTION `cpu_time` RETURN `natural`;

`timing_error` : EXCEPTION;

END `timing`;

A call of the function `cpu_time` returns the CPU-time consumed by the running process in milliseconds. The value `natural'last` will be reached after 24 days, 20 hours, 31 minutes, 23 seconds and 647 milliseconds.

The exception `timing_error` will be raised if a `constraint_error` or `numeric_error` occurs within `cpu_time`.

13.3.3 The Package `COMMAND_ARGUMENTS`

The package `command_arguments` provides a means of accessing within an Ada program the arguments of the command which executed the program. The environment of the process executing the command is also made available. The package specification is:

```
PACKAGE command_arguments IS

  argc : integer;
    -- number of arguments of a command

  FUNCTION argl (nr : natural) RETURN natural;
    -- length of nr-th argument

  FUNCTION argv (nr : natural) RETURN string;
    -- value of nr-th argument

  envc : integer;
    -- number of values in environment vector

  FUNCTION envl (nr : natural) RETURN natural;
    -- length of nr-th value in environment vector

  FUNCTION envv (nr : natural) RETURN string;
    -- value of nr-th value in environment vector

END command_arguments;
```

A call of the function `argv` with actual parameter `nr` delivers the `nr`-th argument ($0 <= nr <= argc - 1$) of the command which executed the Ada program. `argc` returns the number of arguments passed to the command. `argv (0)` returns the name of the command which is currently being executed. Each of the arguments accessed through `argv (nr)` is represented as a string of length `argl (nr)`.

Information about the environment of the process that executes the command may be obtained in an analogous manner. The function `envv (nr)` ($0 <= nr <= envc - 1$) delivers a string of the form "name=value" as specified in *environ*(5). The string length may be obtained by calling `envl (nr)`.

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED
has no effect.

ELABORATE
is fully implemented. The Alsys Ada System assumes a **PRAGMA elaborate**, i.e. stores a unit in the library as if a **PRAGMA elaborate** for a unit *u* was given, if the compiled unit contains an instantiation of *u* (or of a generic program unit in *u*) and if it is clear that *u* must have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a **PROGRAM_ERROR** when elaborating the instantiation.

INLINE
Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for ASSEMBLER and C.

PRAGMA interface (assembler, ...) provides an interface with the internal calling conventions of the Alsys Ada System. See §15.1.3 for further description.

PRAGMA interface (C, ...) is provided to support the C procedure calling standard. §15.1.4 describes how to use this pragma. The subprogram must not be a function returning an unconstrained array type, nor must it have OUT or IN OUT parameters that are not passed by reference. If either of these restrictions is violated, the program is erroneous.

PRAGMA interface should always be used in connection with the **PRAGMA external_name** (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not use names beginning with an underline.

LIST

is fully implemented. Note that a listing is only generated when one of the listing options is specified with the `ada.c` (or `ada.make` or `ada.link`) command.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect; but see also the option `-O` with the `ada.c` command, §4.1.

PACK

see §16.1.

PAGE

is fully implemented. Note that form feed characters in the source do not cause a new page in the listing. They are - as well the other format effectors (horizontal tabulation, vertical tabulation, carriage return, and line feed) - replaced by a ~ character in the listing.

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of the subtype priority, and second, the effect on scheduling (Chapter 14) of not giving this pragma for a task or main program. The range of subtype priority is 0 .. 15, as declared in the predefined library package system (see §15.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA priority for it is the same as if PRAGMA priority (0) had been given (i.e. the task has the lowest priority).

SHARED

is fully supported.

STORAGE_UNIT

has no effect.

SUPPRESS

has no effect, but see §15.1.2 for the implementation-defined PRAGMA suppress_all.

SYSTEM_NAME

has no effect.

15.1.2 Implementation-Defined Pragmas**BYTE_PACK**

see §16.1.

C_CALLABLE (<ada-name>)

this pragma causes the Alsys Ada System to make the subprogram <ada-name> obey the C calling conventions (see PRAGMA interface (C, ...) and §15.1.4), so that the subprogram can be called from a C routine.
See §15.1.5 for further description.

EXTERNAL_NAME (<string>, <ada-name>)

<ada-name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item.

This pragma is used in connection with PRAGMA interface (see §15.1.1) and PRAGMA c_callable (see §15.1.5). If <ada-name> is the name of a subprogram, the Compiler uses the symbol <string> in the call instruction for the subprogram. Furthermore, in connection with the PRAGMA c_callable, the pragma enables the subprogram to be called from a routine written in C using the symbol <string>. The subprogram declaration of <ada-name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.

If <ada-name> is the name of an object, this pragma enables the object to be accessed from outside the Ada program using the symbol <string>, for example from a subprogram written in another language.

Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. The user should not define external names beginning with an underline because Compiler generated names are prefixed with an underline.

RESIDENT (<ada-name>)

this pragma causes the value of the object <ada-name> to be held in memory (rather than in a register) and prevents assignments of a value to the object <ada-name> from being eliminated by the optimizer (see §4.1) of the Alsys Ada Compiler.

PRAGMA resident may be needed to prevent crucial assignments from being eliminated by the optimizer in rare cases in which an object is accessed via its address using the attribute 'address. In all straightforward cases of this sort, i.e. whenever the attribute is used in a way designed to be easily understood by human readers, the Compiler will recognize accesses to the object via 'address as such and will not allow the optimizer to eliminate crucial assignments of values to the object. The following example shows one of the rare cases mentioned above, in which PRAGMA resident is necessary.

In this example, examine_value_at is a non-local procedure with an IN parameter of type system.address, which reads the value at the address passed as actual parameter.

```
WITH system, examine_value_at;
```

```
PROCEDURE example IS
```

```
  ...
  TYPE fcb_type IS ...;
  fcb : fcb_type;
```

```

    fcb_address: system.address;
    ...
    FUNCTION file_control_block_address RETURN system.address IS
    ...
    BEGIN
        IF ... THEN
            RETURN fcb'address;
        ELSE
            ...
        END IF;
    END file_control_block_address;
    ...
BEGIN
    ...
    fcb_address := file_control_block_address; -- may be fcb'address
    fcb := ...;
    examine_value_at (fcb_address);           -- may read fcb
    fcb := ...;
    ...
END example;

```

If this procedure is compiled by the Alsys Ada Compiler without suppression of dead code elimination, i.e. without the -OO option, the first assignment to `fcb` will be eliminated, because the Compiler will not recognize that the value of `fcb` may be read before the next assignment to `fcb`. Therefore

```
PRAGMA resident (fcb);
```

should be inserted after the declaration of `fcb`.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).

SUPPRESS_ALL

causes all the runtime checks described in the LRM (§11.7) to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

15.1.3 Pragma Interface (Assembler, ...)

This section describes the internal calling conventions of the Alsys Ada System, which are the same as those used for subprograms for which `PRAGMA interface (assembler, ...)` is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada; but in object form using the option `-ld` with the `ada.link` (or `ada.c` or `ada.make`) command.

In many cases it is more convenient to follow the C procedure calling standard. Therefore the Alsys Ada System provides `PRAGMA interface (C, ...)`, which supports the standard return of the function result and the standard register saving. This pragma is described in the next section.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The Alsys Ada System uses three different parameter passing mechanisms, depending on the type of a parameter:

- *by value and/or result*: The value of the parameter itself is passed.
- *by reference*: The address of the parameter is passed (like an IN parameter of type `system.address`, which would be passed by value).
- *by descriptor*: A descriptor for the parameter is allocated on the caller's side and is itself passed by reference.

The parameters of a subprogram are passed in registers where possible. The remaining parameters, if any, are passed in an area called a *parameter block*. This area is aligned on a word boundary and contains parameter values (for parameter of scalar types), parameter addresses or descriptor addresses (for parameter of composite types) and alignment gaps.

For a function subprogram an extra register (`$4` or `$f0`) is assigned to contain the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the JAL instruction. The address of the parameter block is passed in `$3`, if necessary. The static link of a subprogram is passed in `$2`, if necessary.

In general, the ordering of the parameter values within the parameter block does not agree with the order specified in the Ada subprogram specification. When determining the position of a parameter within the parameter block, the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are as follows:

Scalar parameters and parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter register or into the parameter block before the call. Then, after the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter register or the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according to their size: A parameter with a size of 8, 16 or 32 bits has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, halfword or word boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching a size specification to the parameter's type in case of an integer, enumeration or fixed point type) it will be byte aligned. Parameters of access types are always aligned to a word boundary.

Parameters of composite types are passed by reference or by descriptor. The descriptors are allocated by the caller and are themselves passed by reference. A descriptor contains the address of the actual parameter object and further information dependent on the specific parameter type. The following composite parameter types are distinguished:

- A parameter of a constrained array type is passed by reference for all parameter modes.
- For a parameter of an unconstrained array type, the descriptor consists of the address of the actual array parameter followed by the bounds for each index range in the array (i.e. FIRST(1), LAST(1), FIRST(2), LAST(2), ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint. This descriptor is itself passed by reference.
- For functions whose return value is an unconstrained array type, a reference to a descriptor for the array is passed in the parameter block as for parameters of mode OUT. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an OUT parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.
- A constrained record parameter is passed by reference for all parameter modes.
- For an unconstrained record parameter of mode IN, the parameter is passed by reference using the address pointing to the record. If the parameter has mode OUT or IN OUT, the value of the CONSTRAINED attribute applied to the actual parameter is passed as an additional boolean IN parameter (which, when not passed in a register, occupies one byte in the parameter block and is aligned to a byte boundary). The boolean IN parameter and the address are treated like

two consecutive parameters in a subprogram specification, i.e. the positions of the two parameters within the parameter block are determined independently of each other.

For all kinds of composite parameter types, the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a word boundary.

Ordering of parameters:

The ordering of the parameters is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function, the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. The registers \$4..\$22 and \$f0..\$f31 are available for parameter passing. A parameter block is only used when there are more parameters than registers of the appropriate class. Registers are used from low numbers to high numbers, the parameter block starts at offset zero and grows to higher offsets. Each parameter is handled as follows:

- A float parameter is allocated the next free even numbered floating point register (the corresponding odd numbered floating point register is not used for parameter passing). If there is no free floating point register, one word is allocated in the parameter block (see below).
- A long_float parameter is allocated the next free floating point register pair. If there is no free floating point register pair, a double word is allocated in the parameter block (see below).
- All other parameters (or their [descriptor] addresses, respectively) are allocated the next free general purpose register from \$4..\$22. If not enough general purpose registers are available for parameter passing, space is allocated in the parameter block depending on the representation of the parameter type (see below).
- If a parameter cannot be passed in a register, space is allocated in the parameter block as follows:

Because of the size and alignment requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block can be larger than the sum of the sizes of all the parameters.

In order to minimize the size of the gaps in a parameter block, an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered

whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, halfword or word boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the parameter register or using the parameter block address incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter register or parameter block. For a parameter of a composite type passed by reference the actual parameter value is accessed indirectly via the parameter address passed in a parameter register or in the parameter block. For a parameter of a composite type passed by descriptor the actual parameter value is accessed via the descriptor whose address is passed in a parameter register or in the parameter block. The descriptor contains a pointer to the actual object. When standard entry code sequences are used within the assembler subprogram (see below), the parameter block address is accessible at address -12(\$30).

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, halfword, word or a multiple thereof, depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs `pack` or `byte_pack` (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor, which is passed by reference.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause. Since the mapping of record types is rather complex, record component clauses should be introduced for each record component if an object of that type is to be passed to a non Ada subprogram in order to ensure correct access to the components.

Saving registers:

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers \$1..\$22, \$24..\$25 will be destroyed by the called subprogram, and therefore saves them of its own accord. The stack pointer \$29 will have the same value after the call as before except for functions returning unconstrained arrays. The stack limit register (\$23) will have the same value after the call as before unless the stack of the main task was extended. If the called subprogram wants to modify further registers it has to ensure that the old values are restored upon return from the subprogram. Note that these register saving conventions differ from the C calling standard.

Finally we give the appropriate code sequences for the subprogram entry and for the return, which both obey the rules stated above.

A subprogram for which PRAGMA interface (assembler, ...) is specified is - in effect - called with the subprogram calling sequence

```

move    $4,...      | assign IN parameters, if any
move    $f0,...
...
jal      <subprogram address>
nop
...
move    ..., $4      | read OUT parameters, if any
...
```

Thus the appropriate entry code sequence is

```

    addiu    $sp,$sp,-12
    sw       $0,0($sp)
    sw       $fp,4($sp)
    sw       $31,8($sp)
    addiu    $15,$sp,-<frame_size-4>
    addiu    $fp,$sp,4
    slt      $1,$23,$15
    bne      $1,$0,L1
    nop
    jal      _EXTSTCK      | Storage check
    move     $24,$15
L1:
    move     $sp,$15
                                     | The field at address -4($fp) is reserved
                                     | for use by the Ada runtime system

```

The return code sequence is then

```

    move     $sp,$fp
    lw       $31,4($sp)
    lw       $fp,0($sp)
    jr       $31
    addiu    $sp,$sp,8

```

15.1.4 Pragma Interface (C, ...)

The Alsys Ada System supports PRAGMA interface (C, ...).

With the help of this pragma *and* by obeying some rules (described below) subprograms can be called which follow the C procedure calling standard. As the user must know something about the internal calling conventions of the Alsys Ada System we recommend reading §15.1.3 before reading this section and before using PRAGMA interface (C, ...).

For each Ada subprogram for which

```
PRAGMA interface (C, <ada-name>);
```

is specified, the body needs and must not be provided in Ada, but in object form using the option `-ld` with the `ada.link` command. The routine implementing the body of the subprogram `<ada-name>` may be written in any language that obeys the C calling conventions - cf. TC/IX Documentation Set, Programmer's Manual, in particular:

- Saving registers
- Calling mechanism
- C stack frame format.

TC/IX system calls or subroutines are allowed too.

The following parameter and result types are supported:

C Type	Ada Type
<code>int</code>	<code>standard.integer</code>
<code>float</code>	<code>standard.float</code>
<code>double</code>	<code>standard.long_float</code>
<code>pointer</code>	<code>system.address</code>

The calling mechanism for all parameter types is call by value. The Type address may serve to implement all kinds of call by references: The user may build all kinds of objects and pass their addresses to the C subprogram or TC/IX system routine.

The name of the routine which implements the subprogram `<ada-name>` should be specified using `PRAGMA external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not define names beginning with an underline.

The following example shows the intended usage of `PRAGMA interface (C, ...)` to call a TC/IX system routine. The given procedure serves to open a file with a fixed name. It is called in the body of the main program.

```
WITH system;
```

```
PROCEDURE unix_call IS
```

```
  read_mode : CONSTANT integer := 8#0#;
```

```

file_name : CONSTANT string := "/ben1/test/f1" & ascii.nul;
PRAGMA resident (file_name);

ret_code  : integer;

use_error : EXCEPTION;

FUNCTION unix_open (path  : system.address;
                   oflag : integer) RETURN integer;
PRAGMA interface (C, unix_open);
PRAGMA external_name ("open", unix_open);

BEGIN
  ret_code := unix_open ('le_name'address, read_mode);
  IF ret_code = -1 THEN
    RAISE use_error;
  END IF;
END unix_call;

```

15.1.5 Pragma C_callable(...)

Like `PRAGMA interface (C, ...)`, `PRAGMA c_callable` enforces the use of C calling conventions (cf. §15.1.4). The body of a subprogram for which

```
PRAGMA c_callable (<ada-name>);
```

is specified must still be provided in Ada. The subprogram can be called from Ada without any restriction; the purpose of this pragma is to enable a C routine (called from the Ada program) to call back an Ada subprogram.

The name of the subprogram which is to be called by C should be specified using the `PRAGMA external_name` (see §15.1.2), and this external name is to be used in the C routine to call back the Ada subprogram. If no external name is specified, the Compiler will generate an internal name that leads to an unsolved reference during linking.

The `PRAGMA c_callable` is allowed at the place of a declarative item of a library package body or library package specification and must apply to a subprogram declared by an earlier declarative item of the same declarative part or package specification. The pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit.

A `c_callable` subprogram must not be a function returning an unconstrained array type, nor must it have `OUT` or `IN OUT` parameters that are not passed by reference.

Furthermore, it must not propagate any exception to its caller because the exception cannot be handled correctly if the caller is a C routine. For this reason, a `c_callable` Ada subprogram does not make a storage check upon subprogram entry (cf. §15.1.3).

If either of these restrictions is violated, the program is erroneous.

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.

If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.

If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.

For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM(§3.5.5(11))) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM(Annex C(13))) by the corresponding upper-case character. For example, `character'image(nul) = "NUL"`.

MACHINE_OVERFLOW

Yields true for each real type or subtype.

MACHINE_ROUNDS

Yields true for each real type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:

If a length specification (**STORAGE_SIZE**, see §16.2) has been given for that type (static collection), the attribute delivers that specified value.

In case of a dynamic collection, i.e. no length specification by **STORAGE_SIZE** given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed. If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:

If a length specification (**STORAGE_SIZE**, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

15.3 Specification of the Package SYSTEM

The package system as required in the LRM (§13.7) is reprinted here with all implementation-dependent characteristics and extensions filled in.

PACKAGE system IS

TYPE designated_by_address IS LIMITED PRIVATE;

TYPE address IS ACCESS designated_by_address;
FOR address's storage_size USE 0;

address_zero : CONSTANT address := NULL;

FUNCTION "+" (left : address; right : integer) RETURN address;
FUNCTION "+" (left : integer; right : address) RETURN address;
FUNCTION "-" (left : address; right : integer) RETURN address;
FUNCTION "-" (left : address; right : address) RETURN integer;

FUNCTION symbolic_address (symbol : string) RETURN address;

SUBTYPE external_address IS STRING;

-- External addresses use hexadecimal notation with characters
-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
-- "7FFFFFFF"
-- "80000000"
-- "8" represents the same address as "00000008"

FUNCTION convert_address (addr : external_address) RETURN address;
-- convert_address raises CONSTRAINT_ERROR if the external
-- address addr is the empty string, contains characters other
-- than '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
-- value cannot be represented with 32 bits.

FUNCTION convert_address (addr : address) RETURN external_address;
-- The resulting external address consists of exactly 8
-- characters '0'..'9', 'A'..'F'.

TYPE name IS (mips_tcix);
system_name : CONSTANT name := mips_tcix;

```
storage_unit : CONSTANT := 8;
memory_size  : CONSTANT := 2 ** 31;
min_int      : CONSTANT := - 2 ** 31;
max_int      : CONSTANT := 2 ** 31 - 1;
max_digits   : CONSTANT := 15;
max_mantissa : CONSTANT := 31;
fine_delta   : CONSTANT := 2.0 ** (-31);
tick         : CONSTANT := 1.0 / 3686400.0;
```

```
SUBTYPE priority IS integer RANGE 0 .. 15;
```

```
TYPE interrupt_number IS RANGE 1 .. 31;
```

```
interrupt_vector : ARRAY (interrupt_number) OF address;
-- The mapping of signal numbers to interrupt addresses is
-- defined by this array.
```

```
sighup      : CONSTANT := 1;
sigint      : CONSTANT := 2;
sigquit     : CONSTANT := 3;
sigill      : CONSTANT := 4;
sigtrap     : CONSTANT := 5;
sigiot      : CONSTANT := 6;
sigabrt     : CONSTANT := sigiot;
sigemt      : CONSTANT := 7;
sigxcpu     : CONSTANT := sigemt;
sigfpe      : CONSTANT := 8;
sigkill     : CONSTANT := 9;
sigbus      : CONSTANT := 10;
sigsegv     : CONSTANT := 11;
sigsys      : CONSTANT := 12;
sigpipe     : CONSTANT := 13;
sigalrm     : CONSTANT := 14;
sigterm     : CONSTANT := 15;
sigusr1     : CONSTANT := 16;
sigusr2     : CONSTANT := 17;
sigchld     : CONSTANT := 18;
sigcld      : CONSTANT := sigchld;
sigxfsz     : CONSTANT := 19;
sigstop     : CONSTANT := 20;
sigststp    : CONSTANT := 21;
sigpoll     : CONSTANT := 22;
sigio       : CONSTANT := 23;
sigurg      : CONSTANT := 24;
sigwinch    : CONSTANT := 25;
sigvtalrm   : CONSTANT := 26;
sigprof     : CONSTANT := 27;
```

```

sigcont    : CONSTANT := 28;
sigttin    : CONSTANT := 29;
sigttou    : CONSTANT := 30;
siglost    : CONSTANT := 31;

non_ada_error : EXCEPTION;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--   illegal instruction encountered
--   error during address translation
--   illegal address

TYPE exception_id IS NEW address;

no_exception_id      : CONSTANT exception_id := NULL;

-- Coding of the predefined exceptions:
--

FUNCTION constraint_error_id RETURN exception_id;
FUNCTION numeric_error_id   RETURN exception_id;
FUNCTION program_error_id   RETURN exception_id;
FUNCTION storage_error_id   RETURN exception_id;
FUNCTION tasking_error_id   RETURN exception_id;

FUNCTION non_ada_error_id   RETURN exception_id;

FUNCTION status_error_id    RETURN exception_id;
FUNCTION mode_error_id      RETURN exception_id;
FUNCTION name_error_id      RETURN exception_id;
FUNCTION use_error_id       RETURN exception_id;
FUNCTION device_error_id    RETURN exception_id;
FUNCTION end_error_id       RETURN exception_id;
FUNCTION data_error_id      RETURN exception_id;
FUNCTION layout_error_id    RETURN exception_id;

FUNCTION time_error_id      RETURN exception_id;

no_error_code      : CONSTANT := 0;

TYPE exception_information
  IS RECORD
    excp_id          : exception_id;
    -- Identification of the exception. The codings of
    -- the predefined exceptions are given above.
    code_addr        : address;

```

```

        -- Code address where the exception occurred. Depending
        -- on the kind of the exception it may be the address of
        -- the instruction which caused the exception, or it
        -- may be the address of the instruction which would
        -- have been executed if the exception had not occurred.
        error_code      : integer;
    END RECORD;

    PROCEDURE get_exception_information
        (excp_info : OUT exception_information);
    -- The subprogram get_exception_information must only be called
    -- from within an exception handler BEFORE ANY OTHER EXCEPTION
    -- IS RAISED. It then returns the information record about the
    -- actually handled exception.
    -- Otherwise, its result is undefined.

    PROCEDURE raise_exception_id
        (excp_id : exception_id);

    PROCEDURE raise_exception_info
        (excp_info : exception_information);

    -- The subprogram raise_exception_id raises the exception
    -- given as parameter. It corresponds to the RAISE statement.

    -- The subprogram raise_exception_info raises the exception
    -- described by the information record supplied as parameter.
    -- In addition to the subprogram raise_exception_id it allows to
    -- explicitly define all components of the exception information
    -- record.

    -- IT IS INTENDED THAT BOTH SUBPROGRAMS ARE USED ONLY WHEN
    -- INTERFACING WITH THE OPERATING SYSTEM.

    TYPE exit_code IS NEW integer;

    error      : CONSTANT exit_code := 1;
    success    : CONSTANT exit_code := 0;

    errno : integer;
    FOR errno USE AT symbolic_address ("errno");
    -- Allows access to the errno set by the last system call, C, or
    -- assembler routine call that was made on behalf of the calling
    -- task.

    PROCEDURE set_exit_code (val : exit_code);

```

```
-- Specifies the exit code which is returned to the
-- operating system if the Ada program terminates normally.
-- The default exit code is 'success'. If the program is
-- abandoned because of an exception, the exit code is
-- 'error'.
```

PRIVATE

```
-- private declarations
```

END system;

15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.6 Expressions in Address Clauses

See §16.5 of this manual.

15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

```
target_type'SIZE > source_type'SIZE
```

15.8 Characteristics of the I/O Packages

The implementation-dependent characteristics of the I/O packages as defined in the LRM(Chapter 14) are reported in Chapter 17 of this manual.

15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM(§13.10.1).

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation` (`release`) on a collection, all following calls of `release` (`unchecked_deallocation`) until the next `reset` have no effect, i.e. storage is not reclaimed.
- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` (resp. `mark` and `release`) before the `reset`.

15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.

16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of the LRM and provide notes for the use of the features described in each section.

16.1 Pragmas

PACK

As stipulated in the LRM (§13.1), this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type `PRAGMA pack` has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

BYTE_PACK

This is an implementation-defined pragma which takes the same argument as the predefined language `PRAGMA pack` and is allowed at the same positions. For components whose type is an array or record type `PRAGMA byte_pack` has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to `PRAGMA pack` all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, `PRAGMA byte_pack` does not effect packing down to the bit level (for this see `PRAGMA pack`).

16.2 Length Clauses

SIZE

For all integer, fixed point and enumeration types the value must be ≤ 32 ;
for float types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway).

for long_float types the value must be $= 64$ (this is the amount of storage which is associated with these types anyway);

for access types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway).

If any of the above restrictions are violated, the Compiler responds with a **RESTRICTION** error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K (+ 2K) bytes if no length clause is given (cf. Chapter 14). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause. Whether a length clause is given or not, an additional 2K bytes are allotted for runtime activities and the total space allotted is not extended dynamically at runtime.

SMALL

There is no implementation-dependent restriction. Any specification for **SMALL** that is allowed by the LRM can be given. In particular those values for **SMALL** are also supported which are not a power of two.

16.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type integer defined in package **standard**.

16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a `RESTRICTION` error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a `RESTRICTION` error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and the difference between the maximum and the minimum sizes of the variant is greater than 32 bytes, and, in addition, if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object. (If the second condition is not fulfilled, the number of bits allocated for any object of the record type will be the value delivered by the size attribute applied to the record type.)
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. LRM(§13.4(8))) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a `RESTRICTION` error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

16.5.1 Interrupts

Under TC/IX it is not possible to handle hardware interrupts directly within the Ada program; all hardware interrupts are handled by the operating system. In TC/IX, asynchronous events are dealt with by signals (cf. *sigvec(2)*). In the remainder of this section the terms *signal* and *interrupt* should be regarded as synonyms.

An address clause for an entry associates the entry with a signal. When a signal occurs, a signal catching handler, provided by the Ada runtime system, initiates the entry call.

By this mechanism, an interrupt acts as an entry call to that task; such an entry is called an *interrupt entry*.

The interrupt is mapped to an *ordinary* entry call. The entry may also be called by an Ada entry call statement. However, it is assumed that when an interrupt occurs there is no entry call waiting in the entry queue. Otherwise, the program is erroneous and behaves in the following way:

- If an entry call stemming from an interrupt is already queued, this previous entry call is lost.
- The entry call stemming from the interrupt is inserted into the front of the entry queue, so that it is handled before any entry call stemming from an Ada entry call statement.

16.5.1.1 Association between Entry and Interrupt

The association between an entry and an interrupt is achieved via an interrupt number (type `system.interrupt_number`), the range of interrupt numbers being 1 .. 31 (this means that 31 single entries can act as interrupt entries). The meaning of the interrupt (signal) numbers is as defined in *sigvec(2)*. A single parameterless entry of a task can be associated with an interrupt by an address clause (the Compiler does not check these conventions). Since an address value must be given in the address clause, the interrupt number has to be converted into type `system.address`. The array `system.interrupt_vector` is provided for this purpose; it is indexed by an interrupt number to get the corresponding address.

The following example associates the entry `ir` with signal `SIGINT`.

```
...  
TASK handler IS  
  
    ENTRY ir;  
  
    FOR ir USE AT system.interrupt_vector (system.sigint);  
END;  
...
```

The task body contains ordinary accept statements for the entries.

16.5.1.2 Important Implementation Information

There are some important facts which the user of interrupt entries should know about the implementation. First of all, there are some signals which the user should not use within address clauses for entries. These signals are SIGFPE, SIGSEGV, SIGBUS, SIGILL, SIGTRAP and SIGALRM; they are used by the Ada Runtime System to implement exception handling and delay statements (SIGALRM). Programs containing address clauses for entries with these interrupt numbers are erroneous.

Moreover, the Debug Runtime System establishes a signal catching handler for the signals SIGUSR1 and SIGQUIT; hence, debugging of any program containing an address clause for an entry with either of these interrupt numbers is restricted.

When the signal SIGUSR1 is used for an interrupt entry, the break-in and connect commands (described in §8.7.2 and §8.5.1 respectively) must not be used as they send the signal SIGUSR1 to the program.

When the signal SIGQUIT is used for an interrupt entry, no program dump can be created by sending this signal to the program. This restriction does not affect the automatic generation of a program dump when the program is abandoned due to an unhandled exception.

In the absence of address clauses for entries, the Ada Runtime System establishes signal catching handlers only for the signals mentioned above, so all other signals will lead to program abortion as specified in the TC/DX documentation.

A signal catching handler for a specific signal is established when a task which has an interrupt entry for this signal is activated. The signal catching handler is deactivated and the previous handler is restored when the task has been completed. Several tasks with interrupt entries for the same signal may exist in parallel; in this case the signal catching handler is established when the first of these tasks is activated, and deactivated when the last of these tasks has been completed.

16.5.1.2 Example With Interrupt Entries

A complete example for an interrupt entry follows at the end of this section. This example can be found in <ADA_dir>/example/interrupt.ada. In that example, a child process is created (by a TC/IX system call *fork(2)*) which executes the *sh(1)* with the command *sleep 10* (cf. *sleep(1)*). A task named *shell_call_handler* defines the entry *handle_sigchld* which is bound to the occurrence of the signal *SIGCHLD*. Once this signal occurs, i.e. the child process dies after about 10 seconds, this interrupt entry is called and handled.

The log of a sample session follows. The lines starting with "\$" are TC/IX commands, all other lines are output.

```
$ ada.c -v interrupt.ada
ALSYS ADA - COMPILER           Control Data 4000/TC/IX   1.83
Library:      /tmp/adalib
Compiling:    /tmp/interrupt.ada
PROCEDURE INTERRUPT_EXAMPLE
*** No Errors found ***
CPU Time used :      4.1  Seconds

$ ada.link -v -o interrupt interrupt_example
ALSYS ADA - COMPLETER           Control Data 4000/TC/IX   1.83
Library:      /tmp/adalib
Completing: INTERRUPT_EXAMPLE._DUR_IO_B00001
SEPARATE PACKAGE BODY _DUR_IO_B00001
*** No Errors found ***
CPU Time used :      1.0  Seconds
ALSYS ADA - LINKER              Control Data 4000/TC/IX   1.83

$ interrupt
At      45538.33142, "sleep 10" started.
At      45549.63086, workload finished
At      45549.63202, "sleep 10" - status: 0

$ cat interrupt.ada
```

```
WITH system, text_io, calendar;

PROCEDURE interrupt_example IS
  PRAGMA priority (1);

  status : integer;
  f : float;
  abort_error : EXCEPTION;

  FUNCTION time_stamp RETURN string IS
    t : calendar.time := calendar.clock;
    result : string (1 .. 15);
    PACKAGE dur_io IS NEW text_io.fixed_io (duration);
  BEGIN
    dur_io.put (result, calendar.seconds (t)); RETURN result;
  END time_stamp;

  PROCEDURE error (func : string) IS
  BEGIN
    text_io.put_line (
      func & ": error" & integer'image (system.errno));
    RAISE abort_error;
  END error;

  PROCEDURE system_call (command : IN string) IS
    result : integer; PRAGMA resident (result);
    arg0 : CONSTANT string := "/bin/sh" & ascii.nul;
    PRAGMA resident (arg0);
    arg1 : CONSTANT string := "sh" & ascii.nul;
    PRAGMA resident (arg1);
    arg2 : CONSTANT string := "-c" & ascii.nul;
    PRAGMA resident (arg2);
    arg3 : string (1 .. 80);
    PRAGMA resident (arg3);
    arg4 : CONSTANT string := " " & ascii.nul;
    PRAGMA resident (arg4);

    FUNCTION unix_fork RETURN integer;
    PRAGMA interface (c, unix_fork);
    PRAGMA external_name ("fork", unix_fork);

    FUNCTION unix_execl (
      arg0, arg1, arg2,
      arg3, arg4 : system.address) RETURN integer;
    PRAGMA interface (c, unix_execl);
    PRAGMA external_name ("execl", unix_execl);
```

```

BEGIN -- system_call
  arg3 (1 .. 1 + command'length - 1) := command;
  arg3 (command'length + 1) := ASCII.nul;
  result := unix_fork;
  IF result = 0 THEN
    -- child process
    DELAY (1.0);
    result := unix_execl (arg0'address,
                        arg1'address, arg2'address,
                        arg3'address, arg4'address);
    error ("execl");
  ELSIF result = -1 THEN
    error ("fork");
  ELSE
    NULL; -- the parent
  END IF;
END system_call;

PROCEDURE wait_for_child (status : OUT integer) IS
  r_status : integer;
  PRAGMA resident (r_status);
  result : integer;

  FUNCTION unix_wait (status : IN system.address) RETURN integer;
    PRAGMA interface (c, unix_wait);
    PRAGMA external_name ("wait", unix_wait);

BEGIN -- wait_for_child
  result := unix_wait (r_status'address);
  IF result = -1 THEN error ("wait"); END IF;
  status := r_status;
END wait_for_child;

TASK shell_call_handler IS
  PRAGMA priority (2);

  ENTRY subshell (command : IN string);
  ENTRY wait (status : OUT integer);
  ENTRY handle_sigchld;
  FOR handle_sigchld USE AT
    system.interrupt_vector(system.sigchld);
END shell_call_handler;

TASK workload IS
  PRAGMA priority (0);
  ENTRY start;
  ENTRY stop;
  ENTRY finished (result : OUT float);
END workload;

```

```

TASK BODY shell_call_handler IS
  last_status : integer := 0;
BEGIN
  LOOP
    SELECT
      ACCEPT subshell (command : IN string) DO
        system_call (command);
      END subshell;
      ACCEPT handle_sigchld;
      workload.stop;
      wait_for_child (last_status);
    OR
      ACCEPT wait (status : OUT integer) DO
        status := last_status;
      END wait;
    OR
      TERMINATE;
    END SELECT;
  END LOOP;
END shell_call_handler;

TASK BODY workload IS
  f : float := 0.0;
BEGIN
  ACCEPT start;
  LOOP
    SELECT
      ACCEPT stop; EXIT;
    ELSE
      IF f >= 10.0 THEN f := 0.0; END IF;
      f := f * 2.0 + 1.0;
    END SELECT;
  END LOOP;
  ACCEPT finished (result : OUT float) DO
    result := f;
  END finished;
END workload;

BEGIN
  shell_call_handler.subshell ("sleep 10");
  text_io.put_line ("At " & time_stamp & ", "sleep 10" started.");
  workload.start;
  workload.finished (f);
  text_io.put_line ("At " & time_stamp & ", workload finished");
  shell_call_handler.wait (status);
  text_io.put_line ("At " & time_stamp & ", "sleep 10" - status:"
    & integer'image (status));
END interrupt_example;

```


16.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

17 Appendix F: I/O

In this chapter we follow the section numbering of Chapter 14 of the LRM and provide notes for the use of the features described in each section.

17.1 External Files and File Objects

An external file is identified by a string that denotes a TC/IX file name. It may consist of up to 1023 characters.

The form string specified for external files is described in §17.2.1.1.

17.2 Sequential and Direct Files

Sequential and direct files are ordinary files which are interpreted to be formatted with records of fixed or variable length. Each element of the file is stored in one record.

In case of a fixed record length each file element has the same size, which may be specified by a form parameter (see §17.2.1.1); if none is specified, it is determined to be $(\text{element_type}'\text{SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$.

In contrast, if a variable record length is chosen, the size of each file element may be different. Each file element is written with its actual length. When reading a file element its size is determined as follows:

- If an object of the `element_type` has a size component (see §16.4) the element size is determined by first reading the corresponding size component from the file.
- If `element_type` is constrained, the size is the minimal number of bytes needed to hold a constrained object of that type.
- In all other cases, the size of the current file element is determined by the size of the variable given for reading.

17.2.1 File Management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in the LRM.

17.2.1.1 The NAME and FORM Parameters

The name parameter must be a TC/IX file name. The function name will return a path name string which is the complete file name of the file opened or created. Each component of the file name (separated by "/") is truncated to 255 characters. Upper and lower case letters within the file name string are distinguished.

The syntax of the form parameter string is defined by:

```
form_parameter ::= [ form_specification { , form_specification } ]
```

```
form_specification ::= keyword [ => value ]
```

```
keyword ::= identifier
```

```
value ::= identifier | numeric_literal
```

For identifier and numeric_literal see LRM(Appendix E). Only an integer literal is allowed as numeric_literal (see LRM(§2.4)). In an identifier or numeric_literal, upper and lower case letters are not distinguished.

In the following, the form specifications which are allowed for all files are described.

```
MODE => numeric_literal
```

This value specifies the access permission of an external file; it only has an effect in a create operation and is ignored in an open. Access rights can be specified for the owner of the file, the members of a group, and for all other users. numeric_literal has to be a three digit octal number.

The access permission is then interpreted as follows:

```
8#400#   read access by owner
8#200#   write access by owner
8#100#   execute access by owner
8#040#   read access by group
..       write/execute access by group, analogously
8#004#   read access by all others
..       write/execute access by others, analogously
```

Each combination of the values specified above is possible. The default value is 8#666#.

The definitive access permission is then determined by the TC/IX System. It will be the specified value for MODE, except that no access right prohibited by the process's file mode creation mask (which may be set by the TC/IX command *umask(1)*) is granted. In other words, the value of each "digit" in the process's file mode creation mask is subtracted from the corresponding "digit" of the specified mode. For example, a file mode creation mask of 8#022# removes group and others write permission (i.e. the default mode 8#666# would become mode 8#644#).

The following form specification is allowed for sequential, direct and text files:

SYNCHRO => OFF | ON | ON_WAIT

It allows reader/writer synchronization of parallel file accesses by different processes, such that only one process may write to a file (and no other process may read from or write to the same file in parallel) or multiple processes may read a file in parallel. This synchronization is achieved through the system call *fcntl(2)*.

By default parallel accesses are not synchronized (SYNCHRO => OFF).

If the form specification SYNCHRO => ON is given, USE_ERROR is raised when the access is not possible (because other processes are accessing the file when write access is requested, or because another process is writing the file when read access is requested). If the form specification SYNCHRO => ON_WAIT is given, the process is blocked when the access is not possible for one of the above reasons. When the access becomes possible, the process is unblocked. USE_ERROR is not raised with SYNCHRO => ON_WAIT.

The following form specification is allowed for sequential and direct files:

RECORD_SIZE => numeric_literal

This value specifies the size of one element on the file (record size) in bytes. This form specification is only allowed for files with fixed record format. If the value is specified for an existing file it must agree with the value of the external file.

By default, $(\text{element_type}'\text{SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$ will be chosen as record size, if the evaluation of this expression does not raise an exception. In this case, the attempt to create or open a file will raise USE_ERROR.

If a fixed record format is used, all objects written to a file which are shorter than the record size are filled up. The content of this extended record area is undefined. An attempt to write an element which is larger than the specified record size will result in the exception *use_error* being raised. This can only happen if the record size is specified explicitly.

17.2.1.2 Sequential Files

A sequential file is represented by an ordinary file that is interpreted to be formatted with either fixed-length or variable-length records (this may be specified by the form parameter).

If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up. The content of this extended record area is undefined.

RECORD_FORMAT => VARIABLE | FIXED

This form specification is used to specify the record format. If the format is specified for an existing file it must agree with the format of the external file.

The default is variable record size. This means that each file element is written with its actual length. A read operation transfers exactly one file element with its actual length.

Fixed record size means that every record is written with the size specified as record size.

APPEND => FALSE | TRUE

If the form specification **APPEND => TRUE** is given for an existing file in an open for an output file, then the file pointer will be set to the end of the file after opening, i.e. the existing file is extended and not rewritten. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value **FALSE** is chosen.

TRUNCATE => FALSE | TRUE

If the form specification **TRUNCATE => TRUE** is given for an existing file in an open for an output file, then the file length is truncated to 0, i.e. the previous contents of the file are deleted. Otherwise the file is rewritten, i.e. if the amount of data written is less than the file size, data previously written will remain at the end of the file. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value **TRUE** is chosen.

The default form string for a sequential file is :

```
"RECORD_FORMAT => VARIABLE, APPEND => FALSE, " &
"TRUNCATE      => TRUE,      MODE   => 8#666# " &
"SYNCHRO      => OFF"
```

17.2.1.3 Direct Files

The implementation dependent type count defined in the package specification of `direct_io` has an upper bound of :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
```

A direct file is represented by an ordinary file that is interpreted to be formatted with records of fixed length. If not explicitly specified, the record size is equal to $(\text{element_type}'\text{SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$.

The default form string for a direct file is :

```
"RECORD_SIZE => ..., MODE => 8#666#, SYNCHRO => OFF"
```

17.3 Text I/O

Text files are sequential character files.

Each line of a text file consists of a sequence of characters terminated by a line terminator, i.e. an ASCII.LF character.

A page terminator is represented by an ASCII.FF character and is always preceded by a line terminator.

A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last character of the file.

Output to a file and to a terminal differ in the following way: If the output refers to a terminal it is unbuffered, which means that each write request in an Ada program

will appear on the terminal immediately. Output to other files is buffered, i.e several characters are saved up and written as a block.

Terminal input is always processed in units of lines.

17.3.1 File Management

Besides the mode specification (cf. §17.2.1.1) the following form specification is allowed:

`APPEND => FALSE | TRUE`

If the form specification `APPEND => TRUE` is given for an existing file in an open for an output file, then the file pointer will be set to the end of the file after opening, i.e. the existing file is extended and not rewritten. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value `FALSE` is chosen.

`NON_BLOCKING => FALSE | TRUE`

If the form specification `NON_BLOCKING => TRUE` is given all I/O operations are non-blocking for the file. If a non-blocking I/O operation cannot be completed immediately the calling task is delayed for a duration specified by the variable `text_io_extension.retry_delay` before the operation is tried again (cf. §13.3.4). By default the value `FALSE` is chosen.

The default form string for a text file is :

```
"APPEND => FALSE,  NON_BLOCKING => FALSE,  " &  
"MODE    => 8#666#,  SYNCHRO    => OFF"
```

17.3.2 Default Input and Output Files

The standard input (resp. output) file is associated with the standard TC/IX files `stdin` resp. `stdout`.

Writing to the TC/IX standard error file `stderr` may be done by using the package `text_io_extension` (cf. §13.3.4). This package may also be used to specify that the operations on one of the three standard files should be non-blocking (cf. §13.3.4).

17.3.3 Implementation-Defined Types

The implementation-dependent types `count` and `field` defined in the package specification of `text_io` have the following upper bounds :

`COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)`

`FIELD'LAST = 512`

17.4 Exceptions in I/O

For each of `name_error`, `use_error`, `device_error` and `data_error` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `io_exceptions` can be raised are as described in LRM(§14.4).

NAME_ERROR

- in an open operation, if the specified file does not exist;
- if the name parameter in a call of the `create` or `open` procedure is not a legal TC/IX file name string; i.e, if a component of the path prefix is not a directory.

USE_ERROR

- whenever an error occurred during an operation of the underlying TC/IX system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a capacity limit is exceeded or for similar reasons;

- if the function name is applied to a temporary file or to the standard input or output file;
- if an attempt is made to write or read to/from a file with fixed record format a record which is larger than the record size determined when the file was opened (cf. §17.2.1.1); in general it is only guaranteed that a file which is created by an Ada program may be reopened and read successfully by another program if the file types and the form strings are the same;
- in a create or open operation for a file with fixed record format (direct file or sequential file with form parameter `RECORD_FORMAT => FIXED`) if no record size is specified and the evaluation of the size of the element type will raise an exception. (For example, if `direct_io` or `sequential_io` is instantiated with an unconstrained array type.)
- if a given form parameter string does not have the correct syntax or if a condition on an individual form specification described in §§17.2-3 is not fulfilled;
- in a create or open operation with form specification `SYNCHRO => CN` when the requested access is currently not possible; see §17.2.1.1 for the exact conditions.

DEVICE_ERROR

is never raised. Instead of this exception the exception `use_error` is raised whenever an error occurred during an operation of the underlying TC/IX system.

DATA_ERROR

the conditions under which `data_error` is raised by `text_io` are laid down in the LRM.

In general, the exception `data_error` is not usually raised by the procedure `read` of `sequential_io` and `direct_io` if the element read is not a legal value of the element type because there is no information about the file type or form strings specified when the file was created.

An illegal value may appear if the package `sequential_io` or `direct_io` was instantiated with a different `element_type` or if a different form parameter string was specified when creating the file. It may also appear if reading a file element is done with a constrained object and the constraint of the file element does not agree with the constraint of the object.

If the element on the file is not a legal value of the element type the effect of reading is undefined. An access to the object that holds the element after reading may cause a `constraint_error`, `storage_error` or `non_ada_error`.

17.5 Low Level I/O

We give here the specification of the package `low_level_io`:

```
PACKAGE low_level_io IS
```

```
    TYPE device_type IS (null_device);
```

```
    TYPE data_type IS
        RECORD
            NULL;
        END RECORD;
```

```
    PROCEDURE send_control    (device : device_type;
                               data    : IN OUT data_type);
```

```
    PROCEDURE receive_control (device : device_type;
                               data    : IN OUT data_type);
```

```
END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.